クラスの概念を利用してSystemVerilogモデルの再利用性を向上

―― 簡易CPUバス・モデルのクラス記述

デバイスの記事

宮下晴信

ここではクラスを利用したSystemVerilogの記述例を紹介する．SystemVerilogは，C++やSystemCと同じようにクラスの概念に対応している．これを利用すると，シミュレーション・モデルやテストベンチの記述量が減る．また，再利用しやすいモデルを作成することができる．（編集部）

SystemVerilogでは，クラス（class）の概念が導入されました．このクラスは，米国Synopsys社の機能検証ツール（テストベンチ開発環境）である「Vera」(1)から取り入れた機能です．クラスの概念は機能の実装（設計記述）には使うことができませんが，モデリングや検証には非常に役に立ちます．すなわち，シミュレーション・モデルやテストベンチの記述量が減ったり，再利用性が向上したりします．

本稿では，前半でSystemVerilogのクラスの基本的な内容について説明し，後半では本誌2004年8月号，pp.80-90の解説記事「SystemVerilogで簡易CPUバス・モデルを記述」で紹介した簡易CPUバス・モデルの一部をクラスを使って実装してみます．

1 SystemVerilogのクラス



まず，SystemVerilogのクラスについて説明していきます．

● C++やSystemCのクラスと若干異なる

SystemVerilogのクラスは，C++やSystemCのクラスとほとんど同じです．すでにこれらの言語を使っていれば，それほど抵抗なく使えると思います．ただし，SystemVerilogのクラスはC++やSystemCのクラスと次の点で異なります．

- **ガベージ・コレクションに対応している**
- **モジュールの中でのみ定義する**
- **メンバ・メソッドで関数（function）とタスク（task）をサポートしている**
- **多重継承をサポートしていない**

ガベージ・コレクションはJavaでもサポートされている機能で，メモリ管理を自動的に行ってくれます．つまり，動的に生成したクラスのインスタンスが不必要になった場合，SystemCのように`delete`関数で明示的にオブジェクト用のメモリを解放する必要がありません．

SystemVerilogのクラスは単独では存在できません．あくまでもモジュール内の関数やタスクと同じように，モジュール内で定義し，定義されたモジュール内でのみ使用できます．ただし，SystemVerilogのクラスを別ファイルで定義し，そのクラスを使うモジュールにおいて，クラスを定義したファイルをインクルードすることができます．こうすることでクラスの再利用を行えます．

SystemVerilogのクラスは，モジュールの関数やタスクと同じように，クラスに関数やタスクをメンバ・メソッドとして定義することができます．また，多重継承は行えませんが，モデリングや検証においてこれはとくに問題となりません．

● new関数でクラスを生成・初期化

SystemVerilogのクラスの簡単な例を**リスト1**に示します．**リスト1**では，`Basket`クラスを定義しています．`Basket`クラスには，3種類のフルーツ，つまりりんご，バナナ，オレンジを入れることができ，三つのメンバ変数（`apple`，`banana`，`orange`）と四つのメンバ関数（`new`，`add`，`del`，`num`）を持っています．クラスのメンバ・メソッドは関数とタスクを持てると説明しましたが，ここでは関数についてのみ説明します．タスクについては，簡易CPUバス・モデルのところで説明します．

SystemVerilogにおいて，クラスの生成および初期化は`new`関数で行うので，すべてのクラスに`new`関数が必要で

リスト1　SystemVerilogのクラス記述

```
typedef enum {APPLE, BANANA, ORANGE} TYPE;

Class Basket;

  int apple;
  int banana;
  int orange;

  function new;
    apple  = 0;
    banana = 0;
    orange = 0;
  endfunction : new

  function void add( TYPE index, int no );
    case( index )
      APPLE  :  apple  += no;
      BANANA :  banana += no;
      ORANGE :  orange += no;
    endcase
  endfunction : add

  function void del( TYPE index, int no );
    case( index )
      APPLE  :  apple  -= no;
      BANANA :  banana -= no;
      ORANGE :  orange -= no;
    endcase
  endfunction : del

  function int num( TYPE index );
    case( index )
      APPLE  :  return apple;
      BANANA :  return banana;
      ORANGE :  return orange;
    endcase
  endfunction : num

endclass : Basket
```

リスト2　Basketクラスの使用例

```
module Test;

`include “Basket.sv”

  Basket basket;

  initial begin

    basket = new;

    if( basket != null ) begin
      $display(“APPLE  = %d”, basket.num( APPLE  ) );
      $display(“BANANA = %d”, basket.num( BANANA ) );
      $display(“ORANGE = %d”, basket.num( ORANGE ) );
      $display(“ ”);

      basket.add( APPLE,  5 );
      basket.add( BANANA, 4 );
      basket.add( ORANGE, 3 );

      $display(“APPLE  = %d”, basket.num( APPLE  ) );
      $display(“BANANA = %d”, basket.num( BANANA ) );
      $display(“ORANGE = %d”, basket.num( ORANGE ) );
      $display(“ ”);

      basket.del( APPLE,  4 );
      basket.del( BANANA, 3 );
      basket.del( ORANGE, 2 );

      $display(“APPLE  = %d”, basket.num( APPLE  ) );
      $display(“BANANA = %d”, basket.num( BANANA ) );
      $display(“ORANGE = %d”, basket.num( ORANGE ) );
      $display(“ ”);
    end

  end
```

す．Basketクラスのnew関数では，三つのメンバ変数を0にしています．SystemVerilogのnew関数は，SystemCのコンストラクタに相当します．

モジュールの関数やタスクと同様に，クラスの関数やタスクにもendfunctionやendtaskの後にラベルを付けることができます．**リスト1**のBasketクラスでも各関数のendfunctionの後にラベルを付けています．

クラスの関数は，モジュールの関数と同じように引き数および戻り値を持つことができます．戻り値には，戻り値がないvoidを使うこともできます．

リスト1のadd，del，num関数の第1引き数のように，typedefで定義した型を使うこともできます[注1]．add，del，num関数の第1引き数の型は，列挙型をtypedefで定義したタイプになり，三つの値（APPLE，BANANA，ORANGE）のみ指定できます．

Basketクラスの各関数は次のようなことをします．add関数は，第1引き数で指定したフルーツを第2引き数で指定した個数分入れます．del関数は，第1引き数で指定したフルーツを第2引き数で指定した個数分取り出します．num関数は，第1引き数で指定したフルーツがいくつあるかの値を返します（ただし，add，del関数については，エラー処理を行っていない）．

● includeディレクティブを使ってクラスを読み込む

リスト2は，Basketクラスの使用例です．**リスト1**のコードをBasket.svというファイルにし，Testモジュール内でincludeディレクティブを使って読み込んでいます．

次にBasketクラスのインスタンスbasketを宣言しています．この時点では，インスタンスbasketはBasketクラスのインスタンスとしては初期化されていません．初期化されるのは，new関数を実行したときです．その代わりに，インスタンスbasketの値はnullに初期化されます．

SystemVerilogでは，クラスのインスタンスを宣言したとき，そのインスタンスはnullに初期化されます．インスタンスをnullと比較することで，new関数が実行され

注1：SystemVerilogでは，typedefによって新しいタイプを定義できるようになった．

リスト3　リスト2のシミュレーション結果

```
APPLE  = 0
BANANA = 0
ORANGE = 0

APPLE  = 5
BANANA = 4
ORANGE = 3

APPLE  = 1
BANANA = 1
ORANGE = 1
```

たかどうかを確認できます．

インスタンスのnew関数以外のメンバ関数へのアクセスは，インスタンスの後にピリオド（ . ）を付け，その後，メンバ関数名，引き数の並びになります．

initialブロック内で，インスタンスbasketをnew関数で初期化し，各フルーツの数をnum関数で表示します．次に，りんご（APPLE）を5個，バナナ（BANANA）を4個，オレンジ（ORANGE）を3個，add関数でインスタンスbasketに入れて，各フルーツの数をnum関数で表示します．そして，りんご（APPLE）を4個，バナナ（BANANA）を3個，オレンジ（ORANGE）を2個，del関数でインスタンスbasketから取り出し，各フルーツの数をnum関数で表示します．

リスト2のシミュレーション結果は**リスト3**のようになります．

リスト4
メンバ変数の実装を変えたクラス記述

```
typedef enum {APPLE, BANANA, ORANGE} TYPE;

class Basket;

  int fruits[*];

  function new;
    fruits[APPLE]  = 0;
    fruits[BANANA] = 0;
    fruits[ORANGE] = 0;
  endfunction : new

  function void add( TYPE index, int no );
    fruits[index]  += no;
  endfunction : add

  function void del( TYPE index, int no );
    fruits[index] -= no;
  endfunction : del

  function int num( TYPE index );
    return fruits[index];
  endfunction : num

endclass : Basket
```

● 連想配列を利用してコード量を減らす

リスト4は，**リスト1**と同じ四つの関数をサポートしていますが，メンバ変数の実装が違います．**リスト1**では三つのint型の変数（apple，banana，orange）ですが，**リスト4**ではint型の連想配列（fruits）です．メンバ変数を連想配列にすることにより，**リスト1**よりコード量が少なくなります．

このようにSystemVerilogのクラスであっても，外部に提供する関数やタスクに戻り値，および引き数を変えないで，内部の実装を変更できます．

Column timeunitとtimeprecision

リストA-1　遅延時間の指定

```
`timescale 1ns/10ps

module A;

 ...

endmodule

`timescale 100ps/1ps

 ...

module B;

 ...

endmodule
```

（**a**）Verilog HDL

```
`timescale 100ps/1ps

module C;
    timeunit      1ns;
    timeprecision 10ps;

 ...

endmodule : C
```

（**b**）SystemVerilog

Verilog HDLでは，VHDLと違って，各モジュール内で遅延時間を指定する際に絶対時間（1nsや1psなど）を指定することができませんでした．各モジュール内の遅延時間の単位および精度は，そのモジュールをコンパイルしたときにtimescaleディレクティブで指定したものとなります．例えば**リストA-1（a）**の場合は，モジュールA内の遅延時間の単位は1nsで，精度は10psになります．また，モジュールB内の遅延時間の単位は100psで，精度は1psになります．

一方，SystemVerilogでは，timeunitとtimeprecisionが追加されました．timeunitとtimeprecisionはモジュールやインターフェース内で使用でき，使用したモジュールやインターフェース内でのみ有効です．timeunitは時間単位を，timeprecisionは時間精度を指定します．例えば，**リストA-1（b）**のモジュールC内では，時間の単位はtimeunitで指定した1nsに，時間の精度はtimeprecisionで指定した10psになります．timescaleディレクティブで指定した値にはなりません．

リスト5
簡易CPUバス・モデルのトップ階層のテストベンチ

```
module Test;

  timeunit      1ns;
  timeprecision 10ps;

  logic         CLK, nRST;

  Clock         clock ( .* );
  Reset         reset ( .* );
  BusInterface bus    ( .* );
  GenericCPU   cpu    ( .* );
  Memory       memory( .* );

endmodule : Test
```

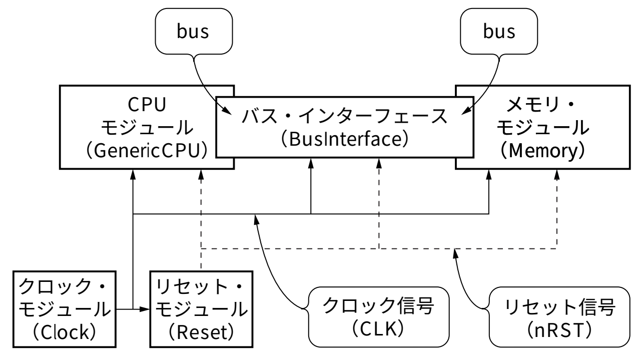


図1　テストベンチの構成
クロック・モジュール(Clock)，リセット・モジュール(Reset)，バス・インターフェース(BusInterface)，CPUモジュール(GenericCPU)，メモリ・モジュール(Memory)，クロック信号(CLK)，リセット信号(nRST)から構成される．CPUモジュールとメモリ・モジュールのポート(bus)は，BusInterfaceに接続している．

❷ クラスを使って簡易CPUバス・モデルを表現

ここまでの説明で，SystemVerilogのクラスの基本について説明しました．ここからは，簡易CPUバス・モデルをSystemVerilogのクラスで記述したものを紹介します．

● トップ階層は五つのモジュールで構成

最初に，簡易CPUバス・モデルのトップ階層のテストベンチを**リスト5**および**図1**に示します．トップ階層のテストベンチのモジュール名はTestで，クロック・モジュール(Clock)，リセット・モジュール(Reset)，バス・インターフェース(BusInterface)，CPUモジュール(GenericCPU)，メモリ・モジュール(Memory)，クロック信号(CLK)，リセット信号(nRST)から構成されます．モジュールおよびインターフェースのポート接続は，.*による接続です．GenericCPUモジュールとMemoryモジュールは，BusInterfaceインターフェースで接続しています．

以降では，Memoryモジュール，GenericCPUモジュールの順に説明します．なお，BusInterfaceインターフェースについては，本誌2004年12月号，pp.121-129に掲載された解説記事「続・SystemVerilogで簡易CPUバス・モデルを記述── DPIを使ってC++モデルを接続する」の**リスト10**を参照してください．また，timeunitとtimeprecisionについては左掲のコラム「timeunitとtimeprecision」を参照してください．

● メモリはlogic型配列からクラスのインスタンスに

まず，BusInterfaceインターフェースによるMemoryモジュールの記述を**リスト6**に示します[注2]．

Memoryモジュールは初期化部(initial文)とメイン部(always文)から構成されます．初期化部は，BusInterfaceインターフェースに対する出力信号bus.READYを‘0’に，双方向のデータ出力信号DRegを32'hzに設定しています．メイン部はステート・マシンになっています．ステート・マシンの各状態は，enum型を使って定義しています．このenum型により，case文の各条件にはS0，S1，S2，S3が使えるようになり，コードも見やすくなります．また，波形表示などでもS0，S1，S2，S3と表示されます．

BusInterfaceインターフェースを使っているので，各信号へのアクセスはbus.Xのようになります．内部メモリ(memory)は32ビットのlogic型で1,024個の配列なので，アドレスbus.Aの下位10ビットのみを使用します．timeunitで1nsを指定しているので，遅延(#delay)は5nsになります．

次に，Memory_ClassクラスによるMemoryモジュールの記述例を**リスト7**に示します．**リスト6**の内部メモリは，logic型の配列からMemory_Classクラスのインスタンスmemになりました．Memory_ClassクラスはMemory_Class.svファイル内で定義しているので，インスタンスmemの前でincludeディレクティブを使ってインクルードしています．

リスト7の初期化部(initial文)では，インスタンスmemをnew関数で生成します．また，メイン・ブロック部

注2：本誌2004年8月号，pp.80-90の解説記事「SystemVerilogで簡易CPUバス・モデルを記述」では，Memoryモジュールのコードについて説明しなかった．

リスト6　BusInterfaceインターフェースによるMemoryモジュールの記述

```
module Memory( BusInterface bus,
                              input logic CLK, nRST );

  timeunit      1ns;
  timeprecision 10ps;

  localparam delay = 5;

  enum { S0, S1, S2, S3 } state;

  logic [31:0] memory[0:1023];

  reg [31:0]     DReg;
  assign bus.D = DReg;

  initial begin
    bus.READY <= 0;
    DReg  <= 32'bz;
  end

  always @(posedge CLK) begin

    if( !nRST ) begin
      state     <=  S0;
      bus.READY <=  #delay 0;
      DReg      <=  #delay 32'bz;
    end
    else begin
      case(state)
        S0 :
          begin
            if( !bus.nCS )
              state <=  S1;
          end
        S1 :
          begin
            if( !bus.nOE )
              DReg  <= #delay memory[bus.A[9:0]];
            state <= S2;
          end
        S2 :
          begin
            if( !bus.nWE )
              memory[bus.A[9:0]] = bus.D;
            bus.READY <= #delay 1;
            state     <= S3;
          end
        S3 :
          begin
            if( bus.nCS ) begin
              state <= S0;
              DReg  <= #delay 32'bz;
            end
            bus.READY <= #delay 0;
          end
      endcase
    end
  end

endmodule : Memory
```

では，インスタンスmemからデータを読み出すときはread関数を，データを書き込むときはwrite関数を使うようにしています．read関数およびwrite関数は，Memory_Classクラスのメンバ関数になります．

リスト8にMemory_Classクラスを示します．Memory_Classクラスには三つの関数（new，read，write）が定義されています．new関数ではとくに何もしていません．read関数では，内部メモリが指定したindex

リスト7　Memory_ClassクラスによるMemoryモジュールの記述

```
module Memory( BusInterface bus,
                              input logic CLK, nRST );

  ...（途中省略）...

  reg [31:0]     DReg;
  assign bus.D = DReg;

`include "Memory_Class.sv"

  Memory_Class mem;

  initial begin

    mem = new;

    bus.READY <= 0;
    DReg  <= 32'bz;
  end

  always @(posedge CLK) begin

  ...（途中省略）...

        S1 :
          begin
            if( !bus.nOE )
              DReg  <= #delay mem.read(bus.A);
            state <= S2;
          end
        S2 :
          begin
            if( !bus.nWE )
              mem.write(bus.A, bus.D);
            bus.READY <= #delay 1;
            state     <= S3;
          end

  ...（途中省略）...

endmodule : Memory
```

の内部メモリの値を返します．write関数は内部メモリが指定したindexの内部メモリにdataを書き込みます．内部メモリのサイズが1,024個であるので，read関数およびwrite関数ではindexの下位10ビットのみを使用しています．

内部メモリのサイズを32ビットに対応するためのコードについては，右掲のコラム「連想配列を使ってMemory_Classクラスを実装する」を参照してください．

● CPUモジュールは2カ所だけ変更

GenericCPUモジュールは，初期化部，メイン部として，命令フェッチ（function [31:0] Fetch），命令デコード（function [31:0] Decode），命令実行タスク（task Execute），内部状態のダンプ（funtion DumpReg），バス・インターフェースとして，ロード・タスク（task Load），ストア・タスク（task Store）から構成されています．なお，SystemVerilog版の簡易CPU

Column 連想配列を使って Memory_Class クラスを実装する

内部メモリのサイズを32ビットに対応させる方法として，連想配列（associative arrays）を使う方法があります．

連想配列では，**リストB-1**のようにサイズを指定する部分を[*]にします．read関数は，memory.exists(index)においてindexで指定したインデックスにすでにデータが書き込まれていたかどうかを調べ，まだ書き込まれていないときはエラー・メッセージを表示し，32'hffffffffを返します．

リストB-1　連想配列の記述例

```
class Memory_Class;

  logic [31:0]  memory[*];

  function new;
  endfunction : new

  function bit [31:0] read( logic [31:0] index );
    if ( memory.exists( index ))
      return memory[index];
    $display( "<<ERROR>> : Unknown Read Data");
    return 32'hffffffff;
  endfunction : read

  function void write( logic [31:0] index,
                                logic [31:0] data );
    memory[index] = data;
  endfunction : write

endclass : Memory_Class
```

リスト8　Memory_Class クラス

```
class Memory_Class;

  logic [31:0]    memory[0:1023];

  function new;
  endfunction : new

  function bit [31:0] read( logic [31:0] index );
    return memory[index[9:0]];
  endfunction : read

  function void write( logic [31:0] index,
                              logic [31:0] data );
    memory[index[9:0]] = data;
  endfunction : write

endclass : Memory_Class
```

バス・モデルの記述については，本誌2004年8月号，pp.80-90の解説記事の**リスト1**および**リスト10**を参照してください．

p.112の**リスト9**にGenericCPU_Classクラスを，p.113の**リスト10**にGenericCPU_Classクラスを使用したGenericCPUモジュールを示します．GenericCPU_Classクラスは，本誌2004年8月号，pp.80-90のSystemVerilogコードに非常に似ていますが，次の点が異なります．

- **初期化部の一部（ROMイメージ・ファイルを読み込む部分）を関数（Load_File）に分離した**
- **メイン部を一つのタスク（Main）にまとめた**

Executeタスク内で呼んでいるLoadタスクおよびStoreタスクは，GenericCPU_Classクラスを使用するモジュール（この場合はGenericCPUモジュール）で定義します．

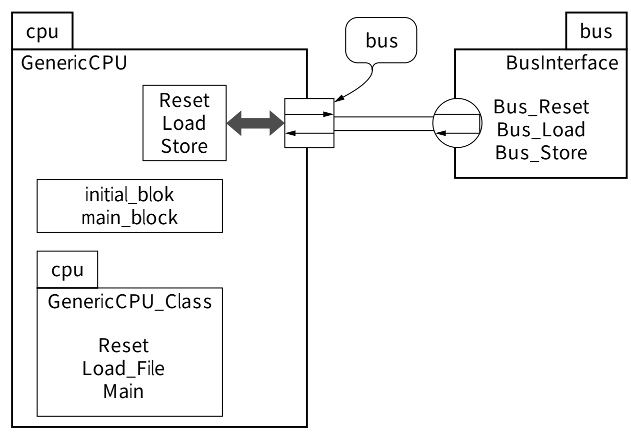


図2　クラスによる簡易CPUバス・モデルの実装

GenericCPUモジュールの三つのタスク（Reset，Load，Store）では，ポートbusのタスク（Bus_Reset，Bus_Load，Bus_Store）を呼んでいる．GenericCPUモジュール内には，GenericCPU_Classクラスのインスタンスcpuがある．

1）GenericCPU_Classクラスを使用する

リスト10のGenericCPUモジュールの記述は，GenericCPU_Classクラスを使うことにより，コード量が非常に少なくなりました．**図2**にGenericCPUモジュールの構成を示します．

初期化部（initial文）では，GenericCPU_Classクラスのインスタンスcpuをnew関数で生成し，cpu.Load_File関数でROMイメージ・ファイルをロードします．そして，Resetタスクを呼びます．Resetタスクでは，cpu.ResetでGenericCPU_ClassクラスのResetタスクを，bus.ResetでBusInterfaceインターフェースのResetタスクを呼びます．

メイン部（always文）では，リセット状態（nRST信号が

リスト9　GenericCPU_Class クラス

```
class GenericCPU_Class;

`include "OPCODE.vh"

  bit [31:0]   memory[0:1023];
  bit [31:0]   Reg[7:0];
  bit          eq, gt, lt;
  int          pc;

  function new;
  endfunction

  function void Load_File( string file );

    $display("¥nGeneric CPU Model
                                  (HDL Model Version)" );
    $display("Copyright (c) 1997 Miya Harayama,
                                  All rights reserved");
    $display("Copyright (c) 2005 Harunobu Miyashita,
                                  All rights reserved");
    $display("               SystemVerilog 3.1a
                             (Class/Interface) Version");
    $display("Internal Memory Size : 1024 Words");

    $display("Loading Internal Memory Data from
                                  '%s' file.", file );
    $readmemh(file, memory, 0);

  endfunction : Load_File

  task Reset;
    for( int i=0 ; i<8 ; i++ )
      Reg[i] =0;
    pc = 0;
    eq = 1'b0;
    gt = 1'b0;
    lt = 1'b0;
  endtask : Reset

  task Main;

    bit [ 7:0]  code;
    bit [ 3:0]  r0, r1;
    bit [15:0]  imm;

    {code, r0, r1, imm} = Decode(Fetch(pc));
    $display( "PC = %x CODE = 0x%x R0 = %d R1
                                        = %d IMM = 0x%x",
              pc, code, r0, r1, imm );
    Execute( code, r0, r1, imm );
    DumpReg;

  endtask : Main

  function bit [31:0] Fetch( int PC );
    return memory[PC];
  endfunction : Fetch

function bit [31:0] Decode( bit [31:0] HEX );
  return {HEX[31:24], HEX[23:20], HEX[19:16], HEX[15: 0]};
endfunction : Decode

task Execute( bit [7:0] CODE, bit [3:0] R0, R1, bit
                                             [15:0] IMM );
  int    jmp;

jmp = 1;
  case(CODE)
    `ADD   : Reg[R0] = Reg[R0] + Reg[R1];
    `SUB   : Reg[R0] = Reg[R0] - Reg[R1];
    `MOV   : Reg[R0] = Reg[R1];
    `SETHI : Reg[R0][31:16] = IMM;
    `SETLO : Reg[R0][15: 0] = IMM;
    `CMP   :
      begin
        if( Reg[R0] == Reg[R1] )
          {eq, gt, lt} = {3'b100};
        else if( Reg[R0] > Reg[R1] )
          {eq, gt, lt} = {3'b010};
        else if( Reg[R0] < Reg[R1] )
          {eq, gt, lt} = {3'b001};
      end
    `JMP   : jmp = {{16 {IMM[15]}}, IMM};
    `JEQ   : if( eq ) jmp = {{16 {IMM[15]}}, IMM};
    `JGT   : if( gt ) jmp = {{16 {IMM[15]}}, IMM};
    `JLT   : if( lt ) jmp = {{16 {IMM[15]}}, IMM};
    `LOAD  :
      begin
        Load( Reg[R0], Reg[R1] );
      end
    `STORE :
      begin
        Store( Reg[R0], Reg[R1] );
      end
    `HALT   : $stop;
    `FINISH : $finish(2);

    default : $display( "ERROR : Generic CPU Model
                  does not support Code 0x%x", CODE );

  endcase

  Reg[0] = 0;
  pc += jmp;

endtask : Execute

task DumpReg;
  $display( "Register Contents¥t¥t¥t¥tROM Contents" );
  for( int i=0, tmp=0 ; i<8 ; i++ ) begin
    tmp = i+pc-3;
    tmp = tmp < 0 ? 0 : tmp;
    $display( "Reg[%d] : %x¥t¥tROM[%x]=%x",
               i, Reg[i], tmp, memory[tmp] ) ;
  end
endtask : DumpReg

endclass : GenericCPU_Class
```

'0')のときにResetタスクを呼び，通常状態のときにcpu.MainでGenericCPU_ClassクラスのMainタスクを呼びます．Mainタスク内では，Fetch関数，Decode関数，Executeタスク，DumpReg関数が呼ばれます．Loadタスク，Storeタスクは，BusInterfaceインターフェースのBus_Loadタスク，Bus_Storeタスクを呼びます．

2）new関数に引き数を追加する

リスト11(a)のように，new関数は引き数を持つことができます．GenericCPU_Classクラスのnew関数に引き数(hex_file)を渡すことにより，**リスト11(b)**のようにcpu生成部とROMイメージ・ファイルのロードを1行で記述することができます．

＊　＊　＊

今年のEDS(Electronic Design and Solution) Fair 2005では，SystemCに加えて，SystemVerilogのフォー

リスト10　GenericCPU_Classクラスを使用したGenericCPUモジュール

```
module GenericCPU( BusInterface bus, input logic CLK, nRST );

  timeunit      1ns;
  timeprecision 10ps;

  parameter string hex_file = "test.hex";

`include "GenericCPU_Class.sv"

  GenericCPU_Class cpu;

  task Reset;

    cpu.Reset();
    bus.Bus_Reset;

  endtask : Reset

  task Load( input bit [31:0] ADDR, output bit [31:0] DATA);

    bus.Bus_Load(ADDR, DATA);

  endtask : Load

  task Store( input bit [31:0] ADDR, input bit [31:0] DATA);

    bus.Bus_Store(ADDR, DATA);

  endtask : Store

  initial begin : initial_block

    cpu = new;

    cpu.Load_File(hex_file);

  Reset;

  end : initial_block

  always @( posedge CLK ) begin : main_block

    if( !nRST )
      Reset;
    else
      cpu.Main;

  end : main_block

endmodule : GenericCPU
```

ラムが開催されました．どちらも立ち見になるほどの盛況でした．SystemCについてはユーザ事例が発表されましたが，SystemVerilogについては，ツールのサポートがまだ十分とは言えないので，チュートリアルのみでした．今後，各EDAベンダのサポートが進めば，ユーザ事例の発表も増えてくると思います．

SystemVerilogのクラスは，本稿で説明した内容のほかに，SystemCと同じように継承(inheritance)や多様性(polymorphism)などの特徴もあります．これらについては，機会をあらためて紹介したいと思います．

リスト11　引き数を持つことができるnew関数

```
function new( string file );

  Load_File( file );

endfunction
```

(**a**) new関数の引き数

```
initial begin : initial_block

  cpu = new(hex_file);

  Reset;

end : initial_block
```

(**b**) new関数の利用例

参考・引用*文献

(1) Faisal Haque, Jonathan Michelson, Khizar Khan；The Art of Verification with Vera, http://www.verificationcentral.com/.
(2) 川北浩孝；システムオンチップ時代のスケーラブル設計手法，各種設計ノウハウ 第4回，pp.207-216，Interface，1997年11月号.
(3) 宮下晴信；抽象度の違いによるモデリング，http://www.systemc.org/projects/sitedocs/document/miyashita_presentation_3/.
(4) 宮下晴信；インターフェースによる抽象度の隠蔽，http://www.systemc.org/projects/sitedocs/document/miyashita_presentation_4/.
(5) SystemVerilog 3.1a Language Reference Manual, 2004, http://www.eda.org/sv/SystemVerilog_3.1a.pdf.
(6) 宮下晴信；特集2 Perlスクリプトでスマートな設計業務，第3章 バス・モデル用フォーマット変換ツール，pp.111-117，Design Wave Magazine，2002年8月号.
(7) 宮下晴信；SystemVerilogで簡易CPUバス・モデルを記述，pp.80-90，Design Wave Magazine，2004年8月号.
(8) 宮下晴信；続・SystemVerilogで簡易CPUバス・モデルを記述──DPIを使ってC++モデルを接続する，pp.121-129，Design Wave Magazine，2004年12月号.
(9) Sutart Sutherland, Simon Davidmann, Peter Flake；SystemVerilog for Design, Springer(旧Kluwer Adademic Publisher), 2004.

みやした・はるのぶ
富士ゼロックス(株)

<筆者プロフィール>
宮下晴信．1991年からVerilog HDLを使い始め，ASIC，FPGA，プリント基板をHDLベースで設計・検証している．今年もSystem Verilog，SystemCともに挑戦していきたいと思っている自称モデリング＆検証エンジニア．